DSA Tutorial Week 2 - Algorithm Analysis

Problem 1 - Algorithm Analysis

Give the tightest possible upper bound for the worst-case runtime for each of the following functions in Big-Oh notation in terms of the variable n. Choose your answer from the following (not given in any order), each of which could be re-used (could be the answer for more than one of a. – d.):

O(1), O(n^2), O(n log n), O(n), O(2^n), O(n^3), O(log n), O(n^4), O(n^5), O(mn)

The correct answer to each can be revealed by selecting the black text below the pointers. (If you printed the tutorial this won't work of course, in that case, godspeed!).

a.

```
void silly(int n, int x, int y) {
    if (x < y) {
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n * i; ++j)
                System.out.println("y = " + y);
   } else {
        System.out.println("x = " + x);
   }
}
```

Pointers:

1. **Nested Loops:** Observe how the loops are nested and how their limits are defined. The innermost loop runs ***n*i*** times, which changes with each iteration of the outer loop.
2. **Growth Rate:** Consider the effect of the outer loop running ***n*** times and the inner loop's limit increasing linearly with each iteration of the outer loop. The multiplication of these two factors will give you the overall growth rate.

Answer : O(n^3)

b.

```
int silly(int n, int m) {
    if (n < 1 || m < 1) return n;
    else if (n < 100)
        return silly(n - m, m);
    else
        return silly(n - 1, m);
}
```

Pointers:

1. **Recursive Calls:** Focus on how the function calls itself recursively and under what conditions. The depth of the recursion is key. Perhaps you can draw a recursion path/tree to find out this depth.
2. **Worst-Case Scenario:** Think about the ***worst-case input*** that would cause the ***maximum number of recursive calls***. This will help identify the upper bound of the runtime.

Answer : O(n)

c.

```
void foo(int number) {
      int steps = 0;
      while (number > 1) {
            number = number / 2;
            steps++;
        }
       System.out.println("Total steps taken to reach 1: " + steps);
}
```

Pointers:

1. **Halving the Problem:** Recognize how the number is being halved in each iteration. This should hint at the logarithmic nature of the problem.
2. **Number of Iterations:** Consider how many times you can divide ***n*** by ***2*** before it becomes ***1*** or less. This directly relates to the logarithmic complexity.

Answer : O(log n)

d.

```
public static int fib(int n) {
      if (n <= 1) {
            return n;
      }
      return fib(n - 1) + fib(n - 2);
}
```

Pointers:

1. **Recursive Pattern:** Look at the structure of the recursive calls. Notice how each call spawns two more calls. Perhaps to understand the problem better, you can draw a recursion tree to analyze how the number of calls increases with each recursive call. Also use this representation to find the total depth of the tree (in terms of ***n***).
2. **Growth:** Think about how the number of calls grows with increasing n. Each step essentially doubles the number of calls, leading to exponential growth.

Answer : $O(2^n)$

e.

```
public static int middle(int[] array) {
    if (array == null || array.length == 0) {
        throw new IllegalArgumentException("Array cannot be null or empty");
    }
    return array[array.length / 2];
}
```

Answer: O(1)

f.

```
public static int[][] pairs(int[] array1, int m, int[] array2, int n) {
    int[][] productPairs = new int[m][n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            productPairs[i][j] = array1[i] * array2[j];
        }
    }
    return productPairs;
}
```

Answer: O(mn)

Reflections

- Reflect on your understanding of Big-Oh notation. How does it help in comparing different algorithms?
- Think about how your approach to analyzing loops has evolved. Are you more comfortable with determining their impact on the overall runtime?
- Reflect on your understanding of recursive functions. How do the number and nature of recursive calls affect the runtime?
- Consider how you managed to simplify and understand the complexity of nested structures. What strategies worked best for you?
- Think about how these skills translate to real-world coding. How might these insights affect your approach to writing and optimizing code?

TEACHER POINTERS

This may be a difficult problem for many students. Feel free to spend more time on this problem and leave one of the other problems for self-study.

- Stress the importance of understanding Big-Oh notation as a way to describe the upper bound of an algorithm's runtime in the worst-case scenario.
- Help students to find the worst-case input if this is important in the analysis.
- Another means of analysis is to write a basic operation count next to each line.
- Students only have to identify where the algorithm's complexity depends on the input time. So constant-time operations can be ignored (unless the run-time is ***O(1)***).
- Guide students to closely examine each loop in the given functions. Emphasize how the number of iterations and nesting of loops affect the overall time complexity.
- For recursive functions, encourage students to consider the depth and nature of recursion. Understanding the pattern and number of recursive calls is crucial.
- Help students learn how to break down complex nested loops and recursive calls into simpler, more understandable components.
- Suggest drawing recursion trees or using tables to visualize how the number of operations grows with each step.

EXAMPLE SOLUTIONS

a. Nested Loops with Linearly Increasing Limit

- **Explanation:** The function silly contains nested loops. The outer loop runs n times, and the inner loop's limit is n * i, which increases linearly with each iteration of the outer loop. This means that

the total number of iterations is a sum of an arithmetic series, leading to the complexity $O(n^2)$. Since this is nested within another loop running n times, the overall complexity is $O(n^3)$.

- **Key Concept:** Understanding the combined effect of nested loops, especially when the inner loop has a limit that increases with each iteration of the outer loop.

b. Recursive Function with Linear Recursion

- **Explanation:** In silly, the recursive calls depend on the value of n. The function keeps reducing n by m or 1 until it becomes less than 1. In the worst-case scenario (when n is significantly larger than m), the number of recursive calls is proportional to n, making the complexity O(n).
- **Key Concept:** Recognizing the linear relationship between the input size and the number of recursive calls in a function.

c. Logarithmic Complexity

- **Explanation:** The foo function reduces the number by half in each iteration of the while loop. This halving continues until the number becomes 1 or less. The number of iterations needed to reduce number to 1 in this manner is logarithmic with respect to number, leading to a time complexity of **O(log n)**.
- **Key Concept:** Understanding the logarithmic nature of algorithms that divide the problem size by a constant factor (in this case, 2) in each step.

d. Exponential Recursive Calls

- **Explanation:** The Fibonacci function `fib` uses recursion where each call generates two more recursive calls. This pattern leads to an exponential increase in the number of calls with respect to n. Drawing a recursion tree for this function shows that the total number of calls approximates 2^n, making the complexity **$O(2^n)$**.
- **Key Concept:** Recognizing the exponential growth in recursive calls, especially in functions where each call leads to multiple subsequent calls.

e. Constant Time Operation

- **Explanation:** The function **`middle`** calculates the middle index of an array and returns the element at that index. This operation is independent of the array's size, as it involves a simple arithmetic calculation and a single array access. Therefore, the time complexity is **O(1)**, or constant time.
- **Key Concept:** Identifying operations that have a constant number of steps regardless of the input size, leading to **O(1)** complexity.

f. Pairwise Product Calculation with O(mn) Complexity

- **Explanation:** The function pairs performs a pairwise multiplication between elements from two **`arrays`**, **`array1`** of size **`m`** and **`array2`** of size **`n`**. It iterates through each element of **`array1`** and, for each of these elements, iterates through each element of **`array2`**, calculating the product. This results in **`m * n`** operations, where each operation is the multiplication of a pair of elements, one from each array. The products are stored in a two-dimensional array **`productPairs`** with dimensions m by n. Since the number of operations directly depends on the sizes of both input arrays, the time complexity is **O(mn)**.
- **Key Concept:** The key concept here is the understanding of how nested loops contribute to the overall complexity of an algorithm. When you have two nested loops, with the outer loop running **`m`** times and the inner loop running **`n`** times, the total number of iterations (and therefore the number of basic operations performed) is the product of these two numbers, **`m * n`**. This scenario is common in algorithms that deal with matrix operations or combinations of elements from two different datasets.

Problem 2 - Find the Duplicates

---

a. Describe an algorithm that checks if an array of integers contains at least ***k*** duplicates of any element. The method should have a time complexity of **O(n^2)** and a storage complexity of **O(1)**. Discuss how the algorithm should work and write it in **pseudocode**.

Pointers:

1. Considering the storage complexity, can you use, for instance: *sets*, *lists*, or any other data structure?
2. Given the ***O(n^2)*** complexity, what do you expect a solution to look like?
3. **Input:** The method should take an array of integers and an integer ***k*** as input.
4. **Output:** Return `true` if any element appears at least ***k*** times, otherwise return false.

Reflections

1. Consider how the loops, each running up to ***n*** times, contribute to the ***O(n^2)*** complexity.
2. Explore different scenarios, such as when ***k*** is larger than the array size or when ***k*** equals 1.
3. Analyze the algorithm's behavior with unique-element arrays versus arrays with multiple duplicates.

A Java example is available in **Duplicates.java**. Though try to write it yourself first.

b. *Improve* the algorithm (written in a.) to check if an array contains at least ***k*** duplicates of any element, but this time with a time complexity of ***O(n)***. This time, you may use ***O(n)*** storage complexity. You will require an extra data structure to solve this problem, when writing the **pseudocode**, you may use this data structure as a given (i.e. you do not have to implement the data structure).

Pointers:

1. Understand how an extra data structure can reduce the need for nested loops.
2. What requirements do we have for this extra data structure? What do we store in it? How about its complexity?
3. Think of how iterating through the array just once (single loop) contributes to achieving the decrease in complexity.

A Java example is available in **Duplicates.java**. Though try to write it yourself first.

Reflections

1. Describe why your algorithm achieves ***O(n)*** time complexity.
2. Think about how your algorithm handles cases such as **k** greater than the array size, **k** equals **1**, and when the array contains all unique elements.

TEACHER POINTERS

Students don't have to write these algorithms in Java. Only in pseudocode.

a.

- The goal is to determine if any integer in the array appears at least k times.
- Since we are limited to ***O(1)*** storage complexity, we cannot use additional data structures like sets or lists. This constraint implies we must solve the problem using the given array itself.
- Given the time complexity of ***O(n²)***, the solution likely involves nested loops, where we compare each element with every other element.
- We should consider edge cases like k being larger than the array's size or k being 1.

b.

- To improve the algorithm to ***O(n)*** time complexity, we can use an additional data structure with ***O(1)*** access time for counting occurrences of each element.
- A suitable data structure for this purpose is a **dictionary or hash table**, where keys are the elements from the array, and values are their respective counts.
- Iterating through the array once and updating the count in the dictionary for each element helps us achieve ***O(n)*** complexity.
- We still need to consider cases like k greater than the array size or when the array contains all unique elements.

EXAMPLE SOLUTION

a.

```
function containsKDuplicates(array, k):
    for i from 0 to length of array:
        count = 0
        for j from 0 to length of array:
            if i != j and array[i] == array[j]:
                increment count
            if count >= k:
                return true
    return false
```

b.

```
function containsKDuplicatesOptimized(array, k):
    define a dictionary countDict
    for each element in array:
        if element is in countDict:
            increment countDict[element]
        else:
            set countDict[element] to 1
        if countDict[element] >= k:
            return true
    return false
```

Problem 3 - Detecting Duplicate Strings

Implement a method that checks for duplicates in an array of strings. The core of this assignment is to create a ***custom hash function*** for strings and use it in a hash set implementation to efficiently track duplicates.

Requirements:

1. **Custom Hash Function:** Design and implement a hash function that converts strings into hash values. Explain your choice of hashing strategy.
2. **Hash Set Creation:** Use your hash function within a hash set structure to manage the strings.
3. **Duplicate Detection:** Traverse the array, using the hash set to detect duplicates.
4. **Input:** An array of strings.
5. **Output:** Return true if any duplicates are found; otherwise, return false.

Start from this code:

```
function customHashFunction(string, size):
    // TODO Implement a simple but effective hash function for strings

function hasDuplicateStrings(array):
    create hashSet using customHashFunction
    for each string in array:
        if hashSet contains hash of string:
            return true
        else:
            add hash of string to hashSet
    return false
```

Start by discussing the task, what is required? What is the problem? How could you approach the task? Write your solution(s) in **pseudocode.**

Pointers:

1. What characteristics make a hash function effective for strings?
2. How does the hash function impact the distribution of strings in the hash set?
3. Consider the role of collision resolution in your hash set and how it affects performance.
4. A naive approach as a starting point, you might consider a simple hash function like summing the **ASCII** values of the characters in the string. Remember, a good hash function for strings should distribute hashes uniformly across a wide range.
5. Research **'string hashing**', **'polynomial rolling hash function'**, and **'hash code for strings'**. These terms will help you understand common methods for converting strings to hash values.

Reflections

- **Understanding Hash Functions:** Reflect on what makes a good hash function for strings. A good hash function should:
    1. Distribute hash values uniformly to minimize collisions.
    2. Be efficient to compute.
    3. Generate different hash values for different strings as much as possible (though collisions are inevitable in practice).

- Understand that **collisions** occur when different strings produce the same hash value. What is your responsibility in dealing with this issue?
- Do you think that different applications require different types of hashing? How about collision resolution?
- If you were to optimize your algorithm, what aspects would you focus on? Would you modify the hash function, change the way collisions are handled, or adjust the hash table size?

TEACHER POINTERS

Understanding the Task

- The task is to detect duplicates in an array of strings using a custom hash function and a hash set.
- Why can we use collisions here to find duplicates? i.e. if we have a deterministic hash function, then any string we hash twice will have the same hash.
- Discuss the importance of hashing in efficiently managing and searching for data, particularly in large datasets.

Designing a Custom Hash Function

- Guide students to consider the characteristics of a good hash function for strings, such as uniform distribution of hash values, minimal collisions, and computational efficiency.
- Suggest starting with a simple approach, like summing ASCII values, but also encourage thinking about its limitations in terms of collision and distribution.
- Introduce more sophisticated string hashing techniques like polynomial rolling hash functions, and discuss why these provide a better distribution of hash values.

Duplicate Detection Algorithm

- Algorithm Walkthrough: Step through the algorithm to detect duplicates. Ensure students understand how the hash set and the hash function work together to identify duplicates.
- Testing and Edge Cases: Discuss the importance of testing with various inputs, including cases with many collisions, to ensure the algorithm's robustness.
- (optional) Motivate students to research hashing and its applications. Terms like 'string hashing', 'polynomial rolling hash function', and 'hash code for strings' can be useful starting points.
- (optional) Discuss real-world scenarios where efficient duplicate detection is crucial, such as in database management or plagiarism detection software.

EXAMPLE SOLUTION

Students do not have to write java for this problem, only pseudocode.

Naive Hash Function

The naive hash function simply adds up the ASCII values of all characters in the string. This method is straightforward but can lead to a high number of collisions, particularly for strings that are permutations of each other (i.e., same characters in a different order).

```
function naiveHashFunction(string, size):
    hash = 0
    for each character in string:
        hash = hash + ASCII value of character
    return hash % size
```

Polynomial Rolling Hash Function

The polynomial rolling hash function calculates the hash of a string by treating each character as a digit in a base *b* number system, where *b* is usually a prime number. This approach considers the position of each character in the string, thereby reducing the likelihood of collisions for different strings.

```
function advancedHashFunction(string, size):
    hash = 0
    prime = 31 // A prime number used as a base
    for each character in string:
        hash = (hash * prime + ASCII value of character) % size
    return hash
```

Polynomial hashing has a rolling property: the fingerprints can be updated efficiently when symbols are added or removed at the ends of the string (provided that an array of powers of ***p modulo M*** of sufficient length is stored). The popular Rabin–Karp pattern matching algorithm is based on this property.

- Polynomial hashing is advantageous due to its rolling property, allowing efficient updating of hash values when characters are added or removed from the string's ends. This is especially useful in algorithms like the Rabin–Karp string search algorithm, which leverages this property for efficient pattern matching.
- The choice of the prime base and the modulo operation plays a crucial role in minimizing collisions and ensuring a more uniform distribution of hash values across the hash table.

Problem 4 - Collision Resolution (Optional Challenge)

The file **HashMap.java** contains a partial implementation of a Dictionary Abstract Data Type (ADT) using a hashmap. However, the implementation lacks a crucial component: collision resolution.

a. Implement **Chaining** for collision resolution:
    i. Make a copy of the original `HashMap.java`.
    ii. Modify and complete the implementation by introducing the chaining method to resolve collisions. Feel free to adapt any part of the existing code to accommodate chaining.
    iii. Specifically, implement the put and get methods to manage collisions using chaining. Ensure each index of the array stores a linked list (or a similar structure) of entries to handle collisions.

b. Implement **Double Hashing** for collision resolution:
    i. Make a copy of the original `HashMap.java`.
    ii. In this new version, replace the chaining method with double hashing for collision resolution. Double hashing involves using a second hash function to determine a new index when a collision happens, aiming to reduce clustering that might occur with chaining.
    iii. Alter the put and get methods to accommodate double hashing. You will also need to implement an additional hash function that differs from the primary one.

Reflection

- **Performance:** How do the two methods behave under heavy load or numerous collisions? Which method handles clustering more effectively?

- **Memory Usage:** Chaining typically requires more memory due to linked list pointers. How does this compare to the potentially increased array size for double hashing?
- **Complexity**: Consider the complexity of implementing and understanding each method. Which method might be more suitable for different types of applications?
- **Use Cases:** In what scenarios might one method be preferred over the other? For instance, chaining might be better for small datasets with few collisions, while double hashing might excel in larger datasets.

Refer to the provided **DoubleHashingHashMap.java** and **ChainingHashMap.java** for sample implementations and additional guidance.

TEACHER POINTERS

This is an optional challenge that students can do. We do not expect that there will be enough time in the tutorial to cover this problem as well.

- Explain that chaining involves creating a linked list at each index of the hashmap's array. When a collision occurs, the new key-value pair is simply added to the list at that index.
- Guide students on how to modify the HashMap.java file to support chaining. This involves changing the array to store linked lists.
- Focus on the put and get methods. In put, students should add the new key-value pair to the list at the calculated index. In get, they need to search through the list at the index to find the key.
- Encourage students to test their implementation with various scenarios, especially cases with multiple collisions.

Implementing Double Hashing:

- Describe double hashing as a technique where a second hash function is used to find another index when a collision occurs.
- Discuss the design of a secondary hash function. It should be independent of the first to ensure a different index is calculated in the event of a collision.
- Instruct students on altering put and get methods to incorporate the second hash function when collisions are detected.
- Encourage testing to observe how double hashing handles clustering compared to chaining.